References

JTpack90: A Parallel, Object-Based, Fortran 90 Linear Algebra Package *

John A. Turner † Robert C. Ferrell ‡ Douglas B. Kothe §

Abstract

We have developed an object-based linear algebra package, currently with emphasis on sparse Krylov methods, driven primarily by needs of the Los Alamos National Laboratory parallel unstructured-mesh casting simulation tool **Telluride**. Support for a number of sparse storage formats, methods, and preconditioners have been implemented, driven primarily by application needs. We describe our object-based Fortran 90 approach, which enhances maintainability, performance, and extensibility, our parallelization approach using a new portable gather/scatter library (**PGSLib**), current capabilities and future plans, and present preliminary performance results on a variety of platforms.

1 Introduction

An effort was initiated recently at Los Alamos National Laboratory (LANL) to build a new 3-D high-resolution tool for simulating casting processes, *i.e.* the flow of molten material into molds and the subsequent cooling and solidification of the material. The simulation process includes incompressible free-surface flow during mold filling, heat transfer-driven convective flows during solidification, and interface physics such as surface tension and phase change, all in complex geometries. This tool is known as **Telluride**, and is described more fully elsewhere in these proceedings [8].

Several decisions were made early in the design stages of **Telluride** which initiated and drove development of **JTpack90**.

- An unstructured-mesh finite-volume approach would be used to the complex geometries that would be modelled.
- Fortran 90 (F90) was chosen as the implementation language. Though this was a fairly risky decision at the time due to the relative scarcity of stable compilers, we felt that the advantages over alternatives warranted the risk. For example, F90 offers numerous syntactic improvements to Fortran 77 (F77), some of which will be discussed later. Compared with C++, we like the fact that arrays are first-class objects in F90, whereas in C++ every code effort seems to have its own array class. Also, the F90 standard had been approved and was in effect. Though this is not as much of a problem with C++ as it was when this effort was started, the standard still has not been finalized. We are now confident that our decision was the correct one, as F90 compilers have become available for virtually all platforms, and the syntactic advantages of F90 have, among other things, allowed

*Supported by the Department of Energy Accelerated Strategic Computing Initiative Program (ASCI)

†Los Alamos National Laboratory (LANL), Transport Methods Group (XTM), MS B226, Los Alamos, NM 87545, *turner@lanl.gov*, http://lune.mst.lanl.gov/\~{}turner

‡Cambridge Power Computing Associates, Ltd., 2 Still St., Brookline, MA 02146, *ferrell@cpca.com*

§LANL, Fluid Dynamics Group (T-3), MS B216, Los Alamos, NM 87545, *dbk@lanl.gov*

us to write code which is maintainable and easy for new members to the team to become productive with.

- Parallelism would be *via* explicit message-passing using MPI (Message Passing Interface). In addition, the message-passing functionality would be encapsulated in an F90-accessible library, thus hiding the details from the higher-level code. This library is known as **PGSLib**, and is described elsewhere in these proceedings [2].
- Robust and efficient solution of large systems of linear equations would be essential, as they would arise in several aspects of a simulation (*e.g.* heat conduction, our projection method for the Navier-Stokes equations, *etc.*).

One of us[1] had developed an F77 package, **JTpack77** [10], which implements a number of Krylov subspace methods for solving linear systems of equations. Since that package was being used successfully for a number of efforts internal to LANL, it made sense to consider it as a candidate from which to build a similar F90 package for the **Telluride** effort (and eventually other applications). In addition, although there are numerous other high-quality packages for iteratively solving systems of linear equations in other languages, such as ITPACK [7] and NSPCG [6] in F77, AZTEC[2] [4] and PETSc[3] [1] in C, and Diffpack[4] in C++, we knew of no such effort in F90. So development of **JTpack90**[5] began with **JTpack77**[6] as a starting point, driven by the needs of **Telluride**[7].

2 Overview of JTpack90

2.1 Design Goals

Although **JTpack90** is a tool driven by the needs of a particular application, we nevertheless wanted to design an infrastructure that would be general enough and flexible enough to allow **JTpack90** to be used by other codes as well. So some of the design goals of **JTpack90** were to provide

- support for basic sparse storage formats within a framework that would allow new formats to be added easily,
- basic solvers and preconditioners, again within a framework that would allow new methods and preconditioners to be added easily,
- support for matrix-free operation (in which the coefficient is not made available to **JTpack90**), and finally,
- support for application-specific preconditioning.

The last two items are particularly important for **Telluride**, since explicit construction of the coefficient is difficult if not impossible for some of the operators in **Telluride**, and since in most cases custom preconditioners based on knowledge of the underlying physics or numerics are far superior to general preconditioners.

2.2 Status

JTpack90 currently provides several basic Krylov subspace iterative solvers (*e.g.* CG, GMRES, TFQMR, *etc.*), along with a number of preconditioners (*e.g.* IC, ILU, multistep weighted Jacobi and SSOR, *etc.*). These are unremarkable, and descriptions

[1] JAT
[2] http://www.cs.sandia.gov/HPCCIT/aztec.html/
[3] http://www.mcs.anl.gov/petsc/petsc.html
[4] http://www.oslo.sintef.no/diffpack/
[5] http://lune.mst.lanl.gov/\~{}turner/JTpack90.html
[6] http://lune.mst.lanl.gov/\~{}turner/JTpack77.html
[7] http://gnarly.lanl.gov/Telluride/Telluride.html

can be found elsewhere (*e.g.* [9]). More interesting is the object-based infrastructure within which these methods and preconditioners are embedded, and which allows easy implementation of new storage formats, methods, and preconditioners.

A great deal of effort has not been expended on general preconditioners, since we knew that we would be using **JTpack90** primarily in matrix-free mode with **Telluride** and would be using preconditioners customized for the physics / numerics at hand.

3 Object-Based Design

Although even **JTpack77** implements a form of object-based design, it is necessarily crude due to the lack of syntactic support for such programming in F77 (for more on how this is accomplished in **JTpack77**, see [10]). F90 provides a number of syntactic advances over F77, including:

- derived types, similar to C structures, which provide user-defined types which may consist of entities of various types,
- generic procedures, which provide polymorphism,
- array-valued functions,
- overloading of intrinsics, and
- modules, which allow encapsulation and access control *via* public and private attributes.

Modules in particular are a powerful addition to the language, and although they can be used in a number of ways, they are often used to group entities that are related in some manner. Two extreme options for a package such as **JTpack90** would be:

- Bundle routines functionally. That is, all routines that perform matrix-vector multiplication for the various sparse storage types could be grouped in a single module. That module would then be used by any routine needing to perform matrix-vector multiplication.
- Use modules to create "classes". That is, bundle a type definition, along with all the routines necessary to perform operations using that type.

In **JTpack90** we have chosen primarily the latter approach. Although all routines in **JTpack90** are encapsulated in modules, there are two primary types of modules.

- "Class" modules each contain a derived-type definition for a sparse storage type, along with all the routines to operate on that type. An edited version of the **JTpack90** module that defines the class for the ELLPACK-ITPACK (ELL) storage format [9] is shown below.

```
module JT_ELL_module
  implicit none
  type JT_ELL_matrix
     real, dimension(:,:), pointer :: values
     integer, dimension(:,:), pointer :: map
  end type JT_ELL_matrix
  interface MatMul
     module procedure Ax
  end interface
  private
  public :: JT_ELL_matrix, MatMul
contains
  function Ax(a,x)
    type(JT_ELL_matrix), intent(in) :: a
```

```
    real, intent(in), dimension(:) :: x
    real, dimension(SIZE(x)) :: Ax
    integer :: j

    Ax = zero
    do j=1,SIZE(a%values, dim=2)
       where (a%map(:,j) /= 0) Ax = Ax + a%values(:,j)*x(a%map(:,j))
    end do

    return
  end function Ax
end module JT_ELL_module
```

The definition of the `JT_ELL_matrix` type appears in the specification portion of the module, and consists of two rank-2 arrays, one real (for the values of the matrix) and one integer (for the column indices).

Only the function for performing matrix-vector multiplication using ELL storage is shown. The real module also contains routines for assigning, loading, dumping, writing, extracting the diagonal from, computing the norm of, computing an incomplete factorization of, *etc.*, an ELL object.

Note that the F90 intrinsic for performing matrix multiplication, `MatMul`, is overloaded by defining a new routine to be used when the arguments match those of the module procedure `Ax`. Note also that `Ax` is an array-valued function, meaning that it returns an array rather than a scalar.

- "Solver" modules each contain all the routines that implement a particular algorithm, such as CG, GMRES, *etc.* A modified version of the **JTpack90** GMRES module is shown below.

```
module JT_GMRES_module
  implicit none
  interface JT_GMRES
     module procedure GMRES_Full
     module procedure GMRES_ELL
  end interface
  private
  public :: JT_GMRES
contains
  subroutine GMRES_Full (status, b, control, x, cpu, &
                         rnormt, errt, rnorm, err, a, ap)
    use JT_Full_module
    real, intent(in), dimension(:,:) :: a
    real, intent(inout), dimension(:,:) :: ap
#include "GMRES-guts.F90"
  end subroutine GMRES_Full
  subroutine GMRES_ELL (status, b, control, x, cpu, &
                        rnormt, errt, rnorm, err, a, ap)
    use JT_ELL_module
    type(JT_ELL_matrix), intent(in) :: a
    type(JT_ELL_matrix), intent(inout) :: ap
#include "GMRES-guts.F90"
  end subroutine GMRES_ELL
end module JT_GMRES_module
```

Two routines are shown, one for standard, full-storage coefficients and one for ELL coefficients. Note that the only difference between the two routines is the

use statement and the declarations of the arrays. Everything else is common between the routines, and is hence pulled out and stored in a common file.

Though in an important sense this shows the significant advances in abstraction allowed by F90, it also illustrates a shortfall, for these cpp acrobatics would not be necessary if F90 had something akin to templates in C++.

Note that this is a hybrid approach in that some of the routines dealing with matrices of a particular storage type reside in the solver modules rather than in the class modules. This was a conscious decision, since it makes adding a new solver much easier at the expense of requiring slightly more effort when adding a new storage type. With our approach, to add a new solver one must simply create a new solver module. Adding a new storage type requires creating a new class module as well as adding a new routine (though really just a "template" with the correct declarations and an include statement) to each of the solver modules.

4 Parallelization *via* PGSLib [2]

Details of our parallelization strategy are given in [8], so we only describe it briefly here. For **JTpack90**, **PGSLib** provides global reduction (*e.g.* dot product, *etc.*) and gather/scatter functionality. The latter is used for the indirect addressing inherent in forming matrix-vector products using sparse storage for matrices. That is, recall the matrix-vector kernel for a matrix stored in ELL format shown previously. Using **PGSLib** this kernel becomes:

```
y = zero
call PGSLib_gather (y, x_pe, ja_pe, ja, trace, mask=(ja_pe /= 0))
y_pe = SUM(a_pe*y, dim=2)
```

where _pe in a variable name denotes the segment of the array local to a particular processor, and `trace` is a **PGSLib** type containing information about how the data is distributed, *etc.*

5 Results

In this section we present parallel results using **Telluride**. Note that although solution of the linear systems represent the majority of time spent in the simulations, all results are for the whole code, not just **JTpack90**. **JTpack90** was operated in matrix-free mode using reverse communication. Global reduction and gather/scatter functionality was provided by **PGSLib**, and no preconditioning is used.

5.1 Implicit Heat Conduction on a Regularly-Connected Mesh

Consider the following conduction test problem, which has an exact analytic solution [5]. Heat is introduced at time zero to an initially cold brick of material on its $y_{\max}$ xz face with a high applied temperature. Both yz faces and the $y_{\min}$ xz face are maintained cold with a low applied temperature, while the two xy faces are insulated. The temperature within the brick attains a steady state distribution, which is computed in **Telluride** by marching the unsteady heat conduction equation forward in time until the temperature distribution does not change. Steady state was attained in these simulations after five time steps, but one large time step, however, could also achieve the desired result since the **Telluride** conduction algorithm is fully implicit. The linear system of equations are solved by **JTpack90** using CG.

First consider a parallel simulation of this problem on a multi-processor shared-memory Digital AlphaServer 8400. Here we partition the brick with a $16 \times 16 \times 192$ mesh that is block decomposed evenly along the z axis, *i.e.* , each processor receives a $16 \times 16 \times N_z$ mesh, where N_z is some subset of the total mesh in the z direction (192). Table 1 displays the excellent parallel efficiencies realized for this problem on

TABLE 1
Implicit heat flow on 16 × 16 × 192 mesh (300 MHz Digital AlphaServer 8400). [a]

Processors	CPU Time (μs/cell/cycle)	Efficiency
1	583	1.00
2	258	1.13
3	162	1.20
4	129	1.13
6	93	1.04
8	69	1.06

[a] `http://www.dec.com/info/alphaserver/products.html`

TABLE 2
Implicit heat flow on 16 × 16 × 320 mesh (67 MHz IBM SP2). [a]

Processors	CPU Time (μs/cell/cycle)	Efficiency
1	1113	1.00
2	635	0.88
10	124	0.90
20	65	0.86

[a] `http://www.rs6000.ibm.com/hardware/largescale/index.html`

this architecture. The superlinear speedups achieved are likely due to cache effects (decreased cache utilization on fewer processors).

Now consider the parallel simulation on a multi-processor distributed-memory IBM SP2. Here we partition the brick with a 16 × 16 × 320 mesh, and again block decompose the mesh evenly along the z axis. The parallel efficiencies, as shown in Table 2, are still quite high, being >85% for all processor numbers tested. This performance is surprising in light of the fact that this mesh is treated as fully unstructured (despite its being simply-connected), necessitating the use of many parallel gather/scatter functions from **PGSLib** [2].

Though these are encouraging results, we emphasize that these represent a rather idealized problem in that the decomposition is optimal. A somewhat more realistic result is given in the next section.

5.2 Solidification on an Unstructured Hex Mesh

Figure 5.2 shows a 6480-element unstructured hex mesh for a part cast for the LANL inertial confinement fusion program. The chalice consists of a hemispherical shell two inches in diameter. The shell is gated at its pole with a cylindrical "hot top" one inch in diameter and about 1.5 inches tall. The hot top serves to continuously supply liquid metal to the hemispherical shell during filling/solidification (to avoid shrinkage defects). The hot top is then cut away and machined after solidification to give the final product (the hemispherical shell). Here the mesh has been decomposed by CHACO [3] for eight

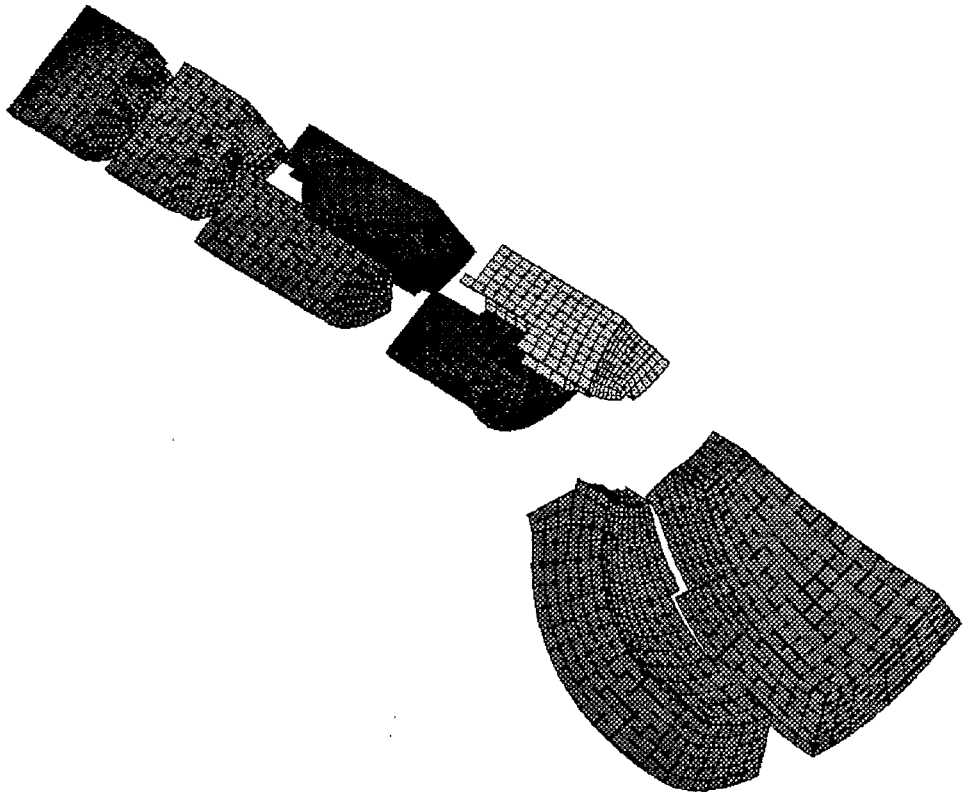

FIG. 1. *The chalice mesh, decomposed for 8 processors by CHACO [3].*

processors.

Although we have also simulated the filling of this mold, we present only solidification results here. For this simulation, the mold cavity is assumed to be initially full of quiescent liquid copper at 1270°C. Because only one 90° quadrant is simulated, elements along the two vertical symmetry planes are assumed insulated. The top horizontal plane of the hot top is assumed insulated because of its proximity (1 inch) to the (hot) crucible. For the inner hemispherical surface (adjacent to the graphite mold), a convective heat transfer boundary condition is applied with a heat transfer coefficient of 25 W/m^2K. For the outer surfaces, a coefficient of 15 W/m^2K is used, which corresponds to experimental values in stationary air.

Table 3 shows results of an implicit heat flow calculation with solidification on a finer mesh than that shown in Figure 5.2, consisting of 46,386 unstructured hex elements, again on a Digital AlphaServer 8400. Again we see excellent parallel efficiencies, which is quite encouraging since this is a more realistic example of the types of parallel casting simulations **Telluride** must perform.

6 Future Work

We have shown that **JTpack90**, in conjunction with **PGSLib**, achieves encouraging parallel efficiencies for both simple and realistic problems within **Telluride**. Nevertheless, much work remains. Preconditioning is one area on which we have only just begun to focus attention. For example, while we have had success using a loosely-converged CG solution of a low-order approximation to the full operator to precondition the CG solutions in our projection flow algorithm, we also want to examine other strategies, especially multilevel approaches.

TABLE 3

Implicit heat flow with solidification on 46,386-cell chalice mesh (300 MHz Digital AlphaServer 8400). [a]

Processors	CPU Time (μs/cell/cycle)	Efficiency
1	5013	1.00
2	2169	1.15
4	1237	1.03
8	721	0.87

[a] `http://www.dec.com/info/alphaserver/products.html`

References

[1] S. Balay, W. Gropp, L. C. McInnes, and B. Smith, *PETSc 2.0 users manual*, Tech. Rep. ANL-95/11, Argonne National Laboratory, Oct 1996.

[2] R. C. Ferrell, J. A. Turner, and D. B. Kothe, *Developing portable, parallel unstructured mesh simulations*, in Eighth SIAM Conference on Parallel Processing for Scientific Computing (this conference), Minneapolis, MN, 1997, SIAM.

[3] B. Hendrickson and R. Leland, *The Chaco user's guide: Version 2.0*, Technical Report SAND94-2692, Sandia National Laboratories, Albuquerque, NM, 1995.

[4] S. A. Hutchinson, J. N. Shadid, and R. S. Tuminaro, *Aztec user's guide (version 1.0)*, Tech. Rep. SAND95-1559, Sandia National Laboratory, 1995.

[5] F. P. Incropera and D. P. De Witt, *Fundamentals of Heat and Mass Transfer*, John Wiley and Sons, NY, 3rd ed., 1990.

[6] D. R. Kincaid, T. C. Oppe, and W. D. Joubert, *An introduction to the NSPCG software package*, Int. J. Num. Meth. Eng, 27 (1989), pp. 589–608.

[7] D. R. Kincaid, J. R. Respess, D. M. Young, and R. G. Grimes, *ITPACK 2C: A FORTRAN package for solving large sparse linear systems by adaptive accelerated iterative methods*, ACM Trans. Math. Software, 8 (1982), pp. 302–322.

[8] D. B. Kothe, R. C. Ferrell, S. J. Mosso, and J. A. Turner, *A high-resolution finite-volume method for efficient parallel simulation of casting processes on unstructured meshes*, in Eighth SIAM Conference on Parallel Processing for Scientific Computing (this conference), Minneapolis, MN, 1997, SIAM.

[9] Y. Saad, *Iterative Methods for Sparse Linear Systems*, PWS Publishing Company, Boston, MA, 1996.

[10] J. A. Turner, *JTpack77 (LA-CC-93-5) - a Fortran 77 collection of linear algebra routines*, Tech. Rep. LA-UR-97-2, Los Alamos National Laboratory, Los Alamos, NM, Jan. 1996.